Machine modeling and loop optimization for horizontal microcoded machines

François Bodin, François Charot

▶ **To cite this version:**

François Bodin, François Charot. Machine modeling and loop optimization for horizontal microcoded machines. [Research Report] RR-1193, INRIA. 1990. inria-00075365

HAL Id: inria-00075365
https://inria.hal.science/inria-00075365

Submitted on 24 May 2006

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

INRIA

UNITE DE RECHERCHE
INRIA-RENNES

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
BP 105
78153 Le Chesnay Cedex
France
Tél. (1) 39 63 55 11

Rapports de Recherche

N° 1193

Programme 2
Structures Nouvelles d'Ordinateurs

MACHINE MODELING AND LOOP OPTIMIZATION FOR HORIZONTAL MICROCODED MACHINES

François BODIN
François CHAROT

Mars 1989

IRISA

INSTITUT DE RECHERCHE EN INFORMATIQUE
ET SYSTÈMES ALÉATOIRES

Campus Universitaire de Beaulieu
35042 - RENNES CÉDEX

Machine Modeling and Loop optimization for Horizontal Microcoded Machines

François Bodin[1]
François Charot

Février 1990
24 pages
Publication interne **n° 517**

Abstract

Long Instruction Word (LIW) architecture exploits parallelism between various functional units. In order to produce efficient code for such an architecture, the microcode compiler will have to expose a relatively large degree of fine grain parallelism and it will have to take into account the fine level characteristics of the architecture. The goal of this paper is to focus on two main aspects of the compilation process for LIW architectures: micromachine modeling and loop optimization. The machine model that has been defined is firstly described. Then a new loop optimization algorithm based on the loop unrolling technique is introduced and compared to the classical software pipelining algorithm. This algorithm differs from the traditional loop unrolling algorithm because the unrolling of the loop is only used to find a cyclic scheduling of the loop, then this scheduling allows a software pipelining to be constructed.

Modélisation de machines et optimisation de boucles pour des architectures à microcode horizontal

Résumé

Une architecture à instruction longue (architecture LIW) a la particularité d'exploiter le parallélisme en contrôlant simultanément plusieurs opérations dans une même instruction. La gestion de ce parallélisme est à la charge d'un compilateur de microcode qui doit prendre en compte les caractéristiques les plus fines de l'architecture. L'objet de ce papier est de présenter deux aspects importants du processus de compilation de microcode pour des architectures LIW : la modélisation de l'architecture et l'optimisation des boucles. Le modèle de machine est tout d'abord décrit. Un nouvel algorithme d'optimisation de boucles utilisant la technique du dépliage est ensuite présenté et comparé à l'algorithme classique de pipeline logiciel. Cet algorithme diffère de l'algorithme de pipeline logiciel car la technique de dépliage est utilisée pour rechercher un ordonnancement cyclique de la boucle. Cet ordonnancement permet alors de construire un pipeline logiciel de la boucle.

[1]Department of Computer Science, Indiana University, Bloomington, Indiana 47405 (USA).

CENTRE NATIONAL DE LA RECHERCHE SCIENTIFIQUE
(L. A. 227)
UNIVERSITÉ DE RENNES 1 I. N. S. A. DE RENNES

INSTITUT NATIONAL DE RECHERCHE
EN INFORMATIQUE ET EN AUTOMATIQUE
(LABORATOIRE DE RENNES)

1 Introduction

To increase performance, RISC architecture relies heavily on technology for achieving short cycle, and on a simplified controlled logic for pipeline (implicit parallelism). In general this requires the compiler to expose a small degree of parallelism: this can be achieve by a post pass optimization on the object code. On the other hand VLIW architecture uses explicit parallelism between various functional units and stresses much more the compiler which has in general to expose a relatively large degree of fine grain parallelism. However the recent trend in RISC design toward multiple instruction issue will force the compiler to find larger and larger degree of parallelism. This requires a much more accurate dependence analysis (this implies propagation of high level dependencies down to the object code) and will have to take into account the fine level characteristics of the architecture. To produce efficient code for such architecture similar techniques as the one developed for VLIW will have to be used.

In this paper, we present a framework to model microcoded LIW architectures and to optimize loops for such architectures. The architecture model can be easily extended to cover architectures with high level instructions set.

The microcode compiler first detects parallelism between instructions, then maps it into wide instruction words to exploit all the parallelism between the various functional units. The mapping itself is decomposed in two parts, the microoperations generation, and optimization process which tries to schedule independent microoperations in the same cycle. In general for arbitrary codes, the optimization can be carried out using two classical approachs: microcode compaction (local optimization inside the basic bloc), or percolation scheduling [19]. For the case of loop structures, specific methods using regularity of the control structure have been devised: software pipelining or loop unrolling.

The quality of the code produced by any of these optimization procedures does not depend only on the method itself but also on the accuracy of the data dependence analysis and the "precision" of the machine description. By "precision" we mean that all the potential parallelism at the hardware level has to be exploitable by the optimizer. This require a very fine level description of the hardware resources and of their usages.

One of the major characteristic of our approach is to use such a fine level description of the architecture. Moreover, this description is a parameter of the optimization procedure, so retargetibily of the optimizer is easily achieved.

A microcode compiler forms a pipeline from the parser through the resource allocators, the code generator, and the microcode optimizer, accepting inputs from the machine description module as appropriate to tailor code for a particular target machine as illustrated in Figure 1. The major modules of such a compiler are now briefly described:

- the *syntax analyzer* translates the source program into an intermediate code (for instance quadruple forms [1]). This phase is machine independent and can be easily updated to enhance the source language. This module computes data dependencies for the optimizer.

- the *microcode generator* translates the intermediate code into sequential microcode. This phase is machine dependent and manages architecture resources like registers and memory banks.

- the *machine description parser* parses the machine description and generates the input to the optimizer. The machine description allows all the features and timing constraints of the microarchitecture to be fully specified. This part is of major importance in order to be able to support any improvement and modification of the hardware and makes the optimizer completely machine independent.

- the *microcode optimizer* parallelizes the sequential microcode and generates horizontal microcode. The optimizer is the most typical part of a compiler for a horizontal microarchitecture. It manages the multiple pipeline functional units and the fine grain parallelism of the machine.

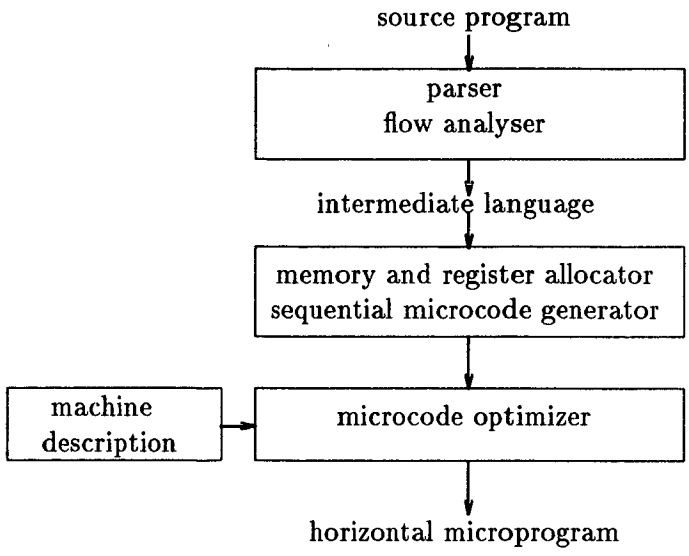


Figure 1: Compiler organization

The goal of the paper is not to deal with the entire compilation process but to focus on two main aspects of this process: micromachine modeling and loop microcode optimization. Section 2 is dedicated to the micromachine modeling. The machine model that has been defined incorporates features as latch and transient resources (i.e. pipeline stage, bus) that can be used explicitly as input and output of a microoperation. These kinds of machine resources introduce special timings constraints for the compaction process. To take into account these timings constraint we introduce, in section 3, the notion of template. The two subsequent sections deal with loop optimization: section 4 gives an overview of the classical software pipelining technique, and section 5 introduces a new loop optimization algorithm based on the loop unrolling technique. This algorithm differs from the traditional loop unrolling algorithm because the unrolling of the loop is only used to find a cyclic scheduling of the loop, then this scheduling allows a software pipelining to be constructed. Finally, the two loop optimization algorithms are compared. The target processor used for this comparison is a LIW processor designed at IRISA[4].

2 Micromachine Modeling

The machine model has to fulfill two criteria:

- flexibility: it has to be general enough to allow retargitibility over a wide range of architecture.
- accuracy: It must be accurate enough to take into account low level features of the machine.

Our model covers architectures with multiple functional units, and various classes of storage devices. However we restrict the target architectures to the ones such that resource behavior is entirely synchronous and predictable. Our model handles architecture features such as:

- transient resources: a resource (for example a bus) is said to be transient if its value is implicitly destroyed after some time, the value has to be used before it is lost.
- multi-cycles microoperations: some operations are not executed in one machine cycle. Therefore, an operation which uses a result produced by such a microoperation has to be delayed.

- delay branches: there are delay branches when the sequencer unit has pipeline stages.

It should be noted that such model introduces temporal restrictions between microoperations, which requires specific solution at the compaction level (c.f. section 3.3). There is a tradeoff between the modeling and the complexity of the compaction process. Very general models such as those used by Mueller and al [18] hand up in making the compaction process extremely difficult.

Machine modeling is composed of two parts: the description of the machine resources which defines the elements of the target architecture and the description of the microoperations which specifies the behavior of the machine.

2.1 Resources

The hardware resources are described at the lowest level, accessible by the microoperations. We distinguish four *types* of elementary resources:

Definition 2.1 *A* **static** *resource is a machine element which stores a data permanently, except if there is a modification operation in the program. A static resource is for instance a memory.*

Definition 2.2 *A* **latch** *resource is a machine element which can be read and written simultaneously. When a write is performed, a read has to be be done in the next cycle, otherwise the data is lost. A latch resource is for instance a pipeline stage.*

Definition 2.3 *A* **transient** *resource is a machine element which stores the data only during the write cycle. A bus is a transient resource.*

Definition 2.4 *A* **field** *resource is a machine element which is used for coding a microoperation.*

The **latch** and **transient** resource types are said to be **volatile.** They are necessary because none of them can be described from the others. This distinction between these first three resources is intended to follow exactly the low level behavior of the architecture.

The optimization module gets all the features of the resources from a description which contains all the resources of the micromachine. For instance a register unit resource declaration is:

```
/* name of the register unit */
STORAGE : RFA
/* resource type */
TYPE : STATIC
/* resource capacity: 128 32-bit words */
CAPACITY : ( 32 , 128 )
ESTO
```

R is the set of the resources of the micromachine and $Type$ is a function which gives the type of the resource:

$$Type : R \rightarrow \{transient, latch, static, field\}$$

2.2 Microoperation

The description must supply all the knowledge on the behavior of the microoperations in order to calculate conflicts on resources and data dependencies. This is done by providing timing information for all the resources. A microoperation (MO_i) is defined by a sextuplet composed of the following fields:

$$<name, input, output, element, field, synchro, transfer>$$

- **name** identifies the microoperation.
- **input, output, element** describe the input resources I_{MO_i}, output resources O_{MO_i}, and elements of the architecture U_{MO_i} which are neither input nor output resources ($I_{MO_i} \subset R$, $O_{MO_i} \subset R$, $U_{MO_i} \subset R$).
- **field** is the set of fields and their values ($C_{MO_i} \subset R \times \mathbf{N}$) for the coding of the microoperation.
- **synchro** is a list of quadruples which describe the temporal use of the resource. They have the following structure:

$$<resource, begincycle, endcycle, R/W>$$

  - **resource** is the name of the resource and must be in another field of the description of the microoperation.
  - **begincycle** and **endcycle** specify the time at which the resource is used and released.
  - the **R/W** attribute indicates how the resource is used (write or read). A resource which cannot be shared is declared with the **W** attribute.

We define the function T_{MO_i}:

$$T_{MO_i} : I_{MO_i} \cup O_{MO_i} \cup U_{MO_i} \cup C_{MO_i} \rightarrow$$

$$\mathbf{N} \times \mathbf{N} \times \{read, write, nil\}$$

which indicates how and at what time a resource is used when the microoperation is initiated.

In order to illustrate these notions, let us consider a 3-stage pipeline ALU resource composed of its two input registers named *aluina, aluinb*, its output register *aluout* and its three internal pipeline registers *alu0, alu1, alu2*, *AluI0I8* is the code operation and *AluCond* the status of the result. *aluina, aluinb, aluout* are **transient** type resources, *alu0, alu1, alu2* are **latch** type resource. Its declaration is given in Figure 2.

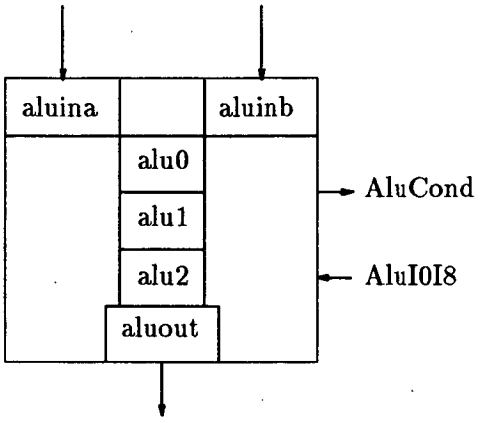
3 Local Compaction

The local compaction algorithm [6] schedules basic blocks of microoperations and tries to minimize the execution time. Dewitt [7] has shown that the problem is NP-complete, because it is similar to the resource constrained scheduling problem [12]. Three compaction algorithms are usually considered:

- the *list scheduling* algorithm which may use a First Come First Serve priority [5]. The microoperations are examined in the original order of the microcode. Each microoperation is scheduled as soon as possible, and when a microoperation is scheduled it is never disallocated.
- the *critical path* algorithm defined by Kleir and Ramamoorthy [15] which identifies critical microoperations and tries to schedule the other without extending the scheduling.
- the *branch and bound* algorithm which guarantees the optimality of the solution since all the solutions are examined, however, the complexity is exponential.

In practice, heuristic algorithms give generally near optimal results. To be correct the algorithm must satisfy the following two conditions:

- *resource constraints*: the scheduling may not introduce conflicts between the functional units of the machine. To avoid conflicts on resources, we use the reservation table formalism [16].
- *Data dependencies and delays.*




```
BMO      : AluOperation, $1
INPUT    : aluina, aluinb
OUTPUT   : aluout, AluCond
ELEM     : alu0, alu1, alu2
FIELDS   : (Alu_I0I8, $1)
TIME     : (alu0, 0, 1, w),   (alu1, 1, 2, w),   (alu2, 2, 3, w),
           (aluina, 0, 1, r), (aluinb, 0, 1, r),
           (aluout, 2, 3, w),
           (AluCond, 2, 3, w)
EMO
```

Figure 2: Declaration of an ALU microoperation

3.1 Data Dependency

In this section we focus our attention on data dependencies for straight line code. The first step of the optimization process consists of constructing the dependency graph. The data dependency is a partial order on microoperations which represents a set of precedence relation to be satisfied in order to preserve the original program semantic. To represent this data dependent relation ($\triangle$), we use an acyclic directed graph (GD) whose nodes are microoperations. Let BB be a basic block of microoperations:

$$BB = MO_1, MO_2, ..., MO_n$$

Let $GD = (S, P)$ be the graph, S the set of vertices labeled with elements of BB and P the set of edges $(MO_i, MO_j) \in S \times S$. Let r be a resource of R. There is an edge between MO_i and MO_j if $MO_i \stackrel{\triangle}{\rightarrow} MO_j$, according to the following rules:

1. MO_i is **direct dependent** on MO_j, noted $MO_i \stackrel{\delta}{\rightarrow} MO_j$, iff:

$$MO_j > MO_i,\ O_{MO_i} \cap I_{MO_j} = \{r\}$$

   and i is the greatest value which satisfies this property for the resource r.

2. MO_i is **output dependent** on MO_j, noted $MO_i \stackrel{\delta^o}{\rightarrow} MO_j$, iff:

$$MO_j > MO_i \; and \; O_{MO_i} \cap O_{MO_j} = \{r\} \; and \; Type(r) = static$$

3. MO_i is **anti dependent** on MO_j, noted $MO_i \stackrel{\overline{\delta}}{\rightarrow} MO_j$, iff:

$$MO_j > MO_i \; and \; I_{MO_i} \cap O_{MO_j} = \{r\} \; and \; Type(r) = static$$

4. $MO_i \stackrel{\Delta}{\rightarrow} MO_j$ if $MO_i \stackrel{\delta}{\rightarrow} MO_j \; or \; MO_i \stackrel{\overline{\delta}}{\rightarrow} MO_j \; or \; MO_i \stackrel{\delta^o}{\rightarrow} MO_j$

The set of volatile resources (definition 2.2 and 2.3) cannot introduce anti dependence or output dependence between microoperations, indeed these dependency types garantee that a value stored in the resource is retained as long as the microoperations which use it are not completed.

3.2 Delay between Microoperations

To be correct, the building of the microprogram must satisfy the delays between microoperations. To deal with delays, edges are valued with a timing pair:

$$(type, d) \subset \{\geq, =\} \times \mathbf{N}$$

this expresses the delay to be satisfied between the two microoperations that are linked by an edge.

If *type* is $\geq$, then the two microoperations must be distant of at least d cycles. If *type* is $=$, then the delay between the microoperations has to be equal to d and the delay is said to be a strict delay. The delay between two microoperations is evaluated depending upon the type of the resource involved in the data dependency and the type of the dependencies. For instance, if MO_j depends on MO_i by a direct dependence ($MO_i \stackrel{\delta}{\rightarrow} MO_j$) and if MO_i is scheduled at cycle c and writes a transient resource r, n cycles later, then MO_j must be scheduled at cycle $c + n$.

The graph GD enhanced with all the delays is called GDD. All the delays are automatically deduced from the description of the microoperations and the resources. Let ω be the function that associates to MO_i the time at which it is initiated. A correct scheduling ω of the microoperations has to satisfy these relations:

- if $MO_i \stackrel{\geq d}{\rightarrow} MO_j$ then $\omega(MO_j) - \omega(MO_i) \geq d$
- if $MO_i \stackrel{=d}{\rightarrow} MO_j$ then $\omega(MO_j) - \omega(MO_i) = d$

3.3 Compaction Algorithm

The compaction algorithm is used to schedule the basic blocks of the microprogram and is a part of the loop scheduling algorithm. The main problem of micromachine modeling is the handling of strict delays, that is those having edges valued with $=$. To deal with this problem, we introduce the notion of **template** which is a set of microoperations connected by strict delays.

The concept of **template** has been used by C. Eisenbeis [8] to name a predefined set of microoperations corresponding to an instruction of the intermediate code of her optimization system. In our case, we use it to point out a set of microoperations, which is dynamically built from the microcode graph. The template can also be modified by the optimization process and is, in our case, independant of the microcode generation. We define a template in the following way:

Definition 3.1 *A* **template** *is a subgraph $B = (S_1, A_1)$ of a dependency graph with delay $GDD = (S, A)$ which is a strongly connected component for the relation $= d$.*

Definition 3.2 *A template B_i is said to be* **legal** *if there exists a scheduling ω_{B_i} that respects delays and data dependencies between the microoperations of the template without conflict on resources.*

By definition, all the templates are disjoint. There exists a single scheduling of the templates since all the microoperations are linked by strict delays. This notion allows the microoperations to be scheduled without the use of a backtracking algorithm. The following condition ensures that the scheduling of the templates exists (sufficient condition):

Property 3.3 *(**Compaction condition**). Let $GB = (EB, AB)$ be the template graph whose nodes are templates. This graph is deduced from the GDD graph. If GB is acyclic and if all the templates are legal then there exists a correct scheduling ω of the microoperations.*

Proof: The proof is based on the use of a list scheduling algorithm on the graph of templates, and the scheduling of microoperations within a template

■

We choose the list scheduling algorithm with the HLF priority. Let $LB = B_1, ..., B_i, ..., B_n$ be the HLF (High Level First) list of templates. The scheduling $\omega_B : LB \rightarrow \mathbf{N}$ is done after the scheduling of the templates. The algorithm is the following:

- Input: the template graph GB.
- Output: the scheduling of the template ω_B.

1. $cc = 0$;
2. **While** $LB \neq \emptyset$
   (a) **For** each template B in LB **Do**
   If B can "be allocated" at cc
   Then schedule B, update the reservation table and LB.
   (b) $cc++$;

By "be allocated", we mean that data dependencies and delays are satisfied and there is no conflict on the resources. The microoperation scheduling ω is then obtained by the function:

$$\omega(MO) = \omega_B(MO \in B_i) + \omega_{B_i}(MO)$$

4 Loop Optimization

Loop optimization is important in order to produce an efficient microcode. It is brought to the fore with the following example which adds two data read from the input queues and then sends the result to the PCS processor output register:

```
for i=1 to 10 do
    read fifo A,read fifo B;
    add;
    write output A
enddo
```

Based on the timing constraints of the architecture, the execution of one iteration is the following:

```
cycle c    : read fifo A, read fifo B
      c+1  : nop
      c+2  : add
      c+3  : nop
      c+4  : nop
      c+5  : write output A
```

Without parallelism between iterations the loop takes 60 cycles to complete. However, an iteration can be initiated every cycle and its execution takes then 15 cycles as illustrated by the following scheme:

```
cycle c    : read fifo A,B
     c+1   : nop            read fifo A,B
     c+2   : add            nop            read fifo A,B
     c+3   : nop            add            nop
     c+4   : nop            nop            add
     c+5   : write output A nop            nop
     c+6   :                write output A nop
     c+7   :                               write output A
```

The parallelization of loops has been approached in different ways. The methods proposed by Touzeau [22] and Eisenbeis [8] are an extension of techniques used to optimize static hardware pipelines [20]. Other algorithms are based on global scheduling algorithms like Trace scheduling [11] or Percolation scheduling [2]. Their principle is first to unroll the loop and then to compact it. The main advantage of such methods is their ability to handle complex loops with branching. However, they do not take into account the cyclic regularity of the loop, and therefore generate great microcode size.

In this section, the software pipelining loop optimization algorithm is described. It uses techniques defined in [8] [22]. We focus our attention on optimizing simple loops; loops made up of assignment statements without branching.

4.1 Data Dependency, V-Loop

The scheduling of a loop must deal with two types of data dependencies:

- the *local data dependencies* (intra-iteration) of the body of the loop (GD).
- the *global data dependencies* of the loop (loop carried dependencies) which give the precedence to the constraints between the microoperations of successive iterations.

The scheduling of a loop is correct only if it respects the two types of dependencies. For instance in a loop containing the statement $a = a + A[i] * B[i]$ a given iteration cannot read the value of a before it has been written by the previous iteration because of the direct data dependency on variable a.

In order to take into account loop carried dependencies the loop body graph GD is enhanced with the global dependencies. The graph is now called GGD. If this graph is acyclic the loop is said to be a vector loop (V-loop [9]) otherwise it is a recurrent loop (R-loop).

4.2 Software Pipelining

The algorithm searches first the scheduling of one iteration, then the final loop is pipelined based on this scheduling. In this paragraph we are only interested in V-loop. The algorithm searches a scheduling which has the following two properties:

- all the iterations have the same scheduling.
- the latency (*initiation interval*) between two successive iterations is L.

This problem can be resumed in searching a global scheduling of the loop Ω that satisfies the data dependencies and is deduced from a local scheduling of the loop body:

$$\Omega(MO_i^j) = j * L + \omega(MO_i)$$

with $MO_i^j, 1 \leq i \leq m, 1 \leq j \leq N$ the microoperation MO_i of iteration j. For a V-loop, a global scheduling of this form satisfies the global dependencies for all positive values of L. With the global scheduling we can construct a new loop, semantically equivalent to the original one. This is called a software pipelining. Figure 4 illustrates the software pipelining execution where the loop body is divided into three parts (C_1, C_2, C_3) concurrently executed. The algorithm consists of two parts:

- Find a scheduling ω. This scheduling must be compatible with the minimum latency L and has to satisfy the data dependencies and the use of resources. In Figure 3, microoperations MO_1, MO_2, MO_3, MO_4 are executed in parallel since successive iterations are initiated with latency L, the amount of resources used by these microoperations may not exceed the machine resources.
- Verify that latency L is compatible with register occupation. If not, there are two solutions. The first one is to try another latency $L + 1$, the second one is to apply techniques like modulo variable expansion [17] or the method defined by Eisenbeis and al [10].

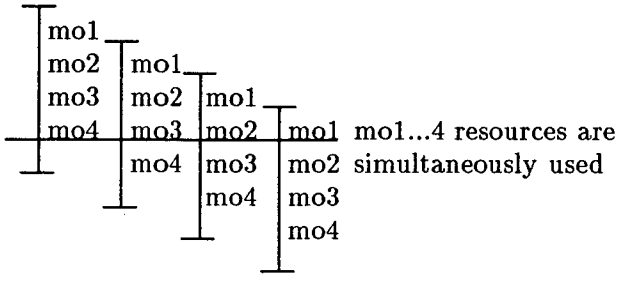


Figure 3: Resource constraints

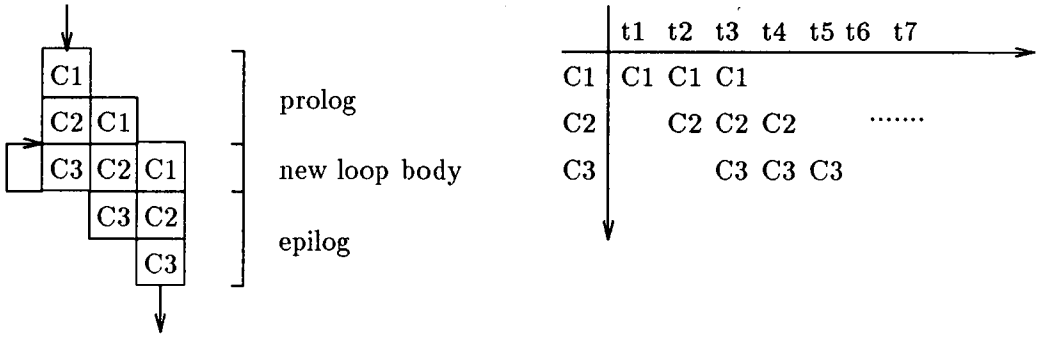


Figure 4: Software pipelining

4.3 Loop Body Scheduling

The algorithm used to find the scheduling of the loop body is similar to the local compaction algorithm, except that:

- The global data dependencies graph GGD is used. It guaranties that loop carried dependencies are respected for all latency values.
- We apply the modulo constraint [21] for latency L on the resource to ensure that the L latency does not generate resource conflict.

Definition 4.1 *Let ω be a scheduling of the loop body C. It is compatible with the* **modulo constraint** *for L if:*

$$\forall k \in \mathbf{Z}, \forall r \in R, \forall\ MO, MO' \in C,\ \ if$$
$$MO \neq MO',\ T_{MO}(r) = (i, j, r/w),\ T_{MO'}(r) = (i', j', r/w)\ then$$
$$[\omega(MO) + i + kL, \omega(MO) + j + kL[\cap[\omega(MO') + i', \omega(MO') + j'[= \emptyset$$

The problem is to limit the latency search space. However, we can restrict this search to an interval defined by the following conditions:

- The lower bound is the minimum number of cycles imposed by the critical resources. A resource is said to be critical if the removal of the contribution of the resource in the lower bound calculation reduces the lower bound.
- The higher bound is the length of the scheduling of the loop body without modulo constraint. If this value is reached there is no parallelism between iterations.

5 Loop Optimization Based on Loop Unrolling

In paragraph 4.2, we have shown how the optimization of V-loops can be treated. This technique is difficult to extend to R-loops. We describe here a new loop optimization algorithm which deals with R-loops in a more natural way than the software pipelining technique does (for instance the method proposed by Lam [17]). The basic idea of the algorithm is to search for a repetitive pattern in the scheduling of the unrolled loop. The compaction algorithm used is quite similar to the one defined previously; when a pattern is found in the scheduling of iterations it is used to construct a software pipelining of the loop. In order to illustrate the method, let us consider the following loop with the hypothesis that only two concurrent memory accesses are authorized without conflict, and that all operations take one cycle to complete:

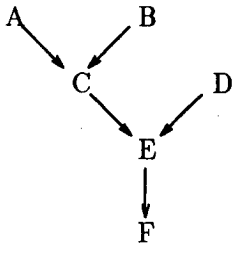
```
for i = 1 to 100 do
{
 t1[i] = t2[i] + t3[i]*t4[i];
}
```

If we unroll and compact successively several iterations of the loop, then we always obtain the same scheduling of the iterations after an unrolling of degree 2 (number of unrolling), as shown in Figure 5 where successive iterations are listed separately, side by side. The vertical axis is time; all statements on a horizontal line are simultaneously executed. Then, we can construct a new loop with two cycles latency between successive iterations.

Aiken and Nicolau [3] have proposed a similar approach which also uses a greedy scheduling and unrolling algorithm. Their algorithm gives an optimal scheduling, but only considers data dependencies. If resources are considered the problem is NP-complete [12]. The algorithm we propose takes into account resources but it does not necessarily give an optimal scheduling. In our view, in the context of microcoded machine, it is of major importance to take into account resources in the scheduling.

One interesting idea of this algorithm is the use of the unrolling to find a cyclic scheduling of the loop. The algorithm permits to construct a software pipelining which is equivalent to a complete unrolling of the loop except for the first iterations. Another idea of the algorithm is to extend the space of solutions for the scheduling of loops on the basis of classical software pipelining techniques. The software pipelining technique asserts that all the iterations have the same scheduling, our algorithm enables several iterations to have different scheduling. The interest of such a method is intrinsic because no "artificial constraint" is introduced in the search of a periodic pattern of a loop scheduling

A = read t3[i]
B = read t4[i]
C = *
D = read t2[i]
E = +
F = write t1[i]



the dependency graph

	1	2	3	4
1	A B			
2	C D	A		
3	E	B D		
4	F	C	A	
5		E	B D	
6		F	C	A
7			E	B D
8			F	C
9				E
10				F

code after 4 iterations

Figure 5: Example of loop unrolling

but the main problem is to find this repetitive pattern after a small number of unrolling. To answer this question, the algorithm has been checked on two sets of benchmarks: a Livermore benchmark, and a set of loops used in the PCS processor specific applications.

5.1 Basis of the algorithm

The purpose of this section is to explain the hypothesis which guarantee the periodicity of the scheduling of an infinite unrolling/compaction process. Let us define some notations first.

Denote C the loop body defined as a ordered sequence of microoperations:

$$C = MO_1, ..., MO_m$$

We define the unrolling function u as:

$$u^i(C) = C_1 C_2 ... C_i$$

where i is the degree of unrolling and the C_j are the successive occurrences of the loop body, and we note:

$$MO_1^i, ..., MO_m^i \in C_i$$

the microoperations of the ith occurrence of the loop body of $u^j(C), j \geq i$.

Let $\Omega_{u^\infty(C)}$ be the scheduling of an infinite degree unrolled loop. $\Omega_{u^i(C)}$ is the scheduling of the loop with an unrolling of degree i. This is a restriction of $\Omega_{u^\infty(C)}$ for the first i iterations. The scheduling of iteration i is noted ω_{C_i} :

$$\omega_{C_i} : MO_1^i, ..., MO_m^i \longrightarrow N$$

with

$$\Omega_{u^i(C)} = \omega_{C_1} \cup \omega_{C_2} \cup ... \cup \omega_{C_i}$$

To prove the periodicity of the scheduling of the infinite unrolled loop we suppose some properties. The **first hypothesis** is about the compaction process. It must be deterministic and *bounded* for the iterations.

Definition 5.1 *The function* $last\_cycle(\omega_{C_i})$ *is equal to the last cycle used by the iteration* C_i*, it has the form:*

$$last\_cycle(\omega_{C_i}) = max\{n + d(MO_j^i) \in N,$$
$$exists MO_j^i \in C_i, \omega_{C_i}(MO_j^i) = n\}$$

where $d(MO)$ *is the duration of microoperation* MO*.*

Definition 5.2 *The function* $first\_cycle(\omega_{C_i})$*is equal to the first cycle used by the iteration* C_i*, it has the form:*

$$first\_cycle(\omega_{C_i}) = min\{n \in N, \exists MO_j^i \in C_i, \omega_{C_i}(MO_j^i) = n\}$$

Definition 5.3 *The* **duration** *of a scheduling* ω_{C_i}*,* $Duration(\omega_{C_i})$*, is the length of the scheduling of iteration* C_i*:*

$$Duration(\omega_{C_i}) = last\_cycle(\omega_{C_i}) - first\_cycle(\omega_{C_i})$$

The iteration scheduling is bounded if and only if the **duration** of the iteration scheduling, ω_{C_i}, is overvalued by a constant S, that is:

$$\exists\, S, \forall\, i \in N \mid Duration(\omega_{C_i}) < S$$

In the following S is called **span** of the iteration scheduling.

The iteration scheduling is done sequentially: the scheduling of iteration $i + 1$ starts when iteration i has been completely scheduled. Moreover, the scheduling of iteration i cannot modify the scheduling of previous iterations. We also suppose another property which limits resource interactions between iterations:

Property 5.4 *The global scheduling of the iterations is such as:*

$$\forall\, 0 < i < j \mid \omega_{C_i} < \omega_{C_j} \iff last\_cycle(\omega_{C_i}) < last\_cycle(\omega_{C_j})$$

If the compaction process is not bounded, a **gap** can appear in the scheduling of the iterations, as shown in Figure 6, for the following loop:

```
for i = 1 to 100 do
{
   x = x + t1[i]*t2[i];
}
```

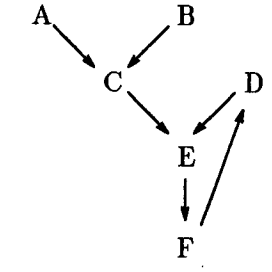
The notion of **gap** is similar to the one exposed by Aiken and Nicolau [3], but if we consider the resources, the gaps in the scheduling cannot be removed without introducing resource conflicts. The **span** limits the growth of gaps.

The **second hypothesis** is about global data dependencies. We suppose that loop carried dependencies only exist between two adjacent iterations, and that they are similar for all pairs of iterations C_iC_{i+1}. The dependency graph considered by the compaction process is the graph of dependencies on the unrolled loop. Loop carried dependencies connect microoperations of successive iterations as illustrated by the graph of the previous loop in Figure 7.

The **third hypothesis** is about resources which are supposed to be limited. Property 5.4 on the order of the global scheduling allows us to bound the machine state according to ω_{C_i}.

Definition 5.5 *The* **machine state** $MS(\Omega_{u^i(C)})$ *associated to iteration* C_i *after the scheduling of the first* i *iterations is a subset of the machine state of* $\Omega_{u^i(C)}$ *which associates to every resource the way it is used at every cycle. The machine state considered for an iteration is limited within the interval:*

$$[last\_cycle(\omega_{C_i}) - S, last\_cycle(\omega_{C_i})]$$



the dependency graph

	1	2	3	4
1	A B			
2	C D	A B		
3	E	C	A B	
4	F	–	C	A B
5		D	–	C
6		E	–	–
7		F	–	–
8			D	–
9			E	–
10			F	–
11				D
12				E
13				F

gap

code after 4 iterations

Figure 6: Gap in the scheduling

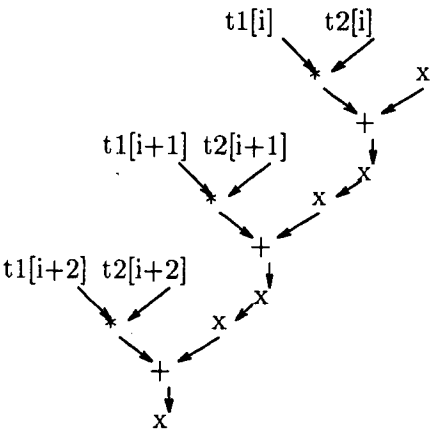


Figure 7: Dependency graph for the unrolled loop

We say that two iterations have the same machine state if they use the resources in the same way with a translation of c cycles:

$$\exists c \in Z,\ \forall r \in R\ \forall x\ \ MS(r, x+c) = MS(r, x)$$

where $MS(r, x)$ is the use of the resource r at cycle c. Two identical machine states are noted:

$$MS(\Omega_{u^i(C)}) \equiv MS(\Omega_{u^j(C)})$$

The machine state can be represented by a reservation table. Figure 8 shows the machine state for the third iteration when S equal ten (the following resources are considered: two memory units MEM1, MEM2, an alu ALU, a multiplier MPY and a register unit RFA).

	MEM1	MEM2	ALU	MPY	RFA
1	X	X			
2	X	X		X	X
3	X	X	X	X	
4				X	X
5					X
6			X		
7					X
8					X
9			X		
10					X

Figure 8: Machine state associated with the third iteration with S = 10

The hypothesis we made are not really restrictive. The hypothesis about the iteration scheduling (the first one) is the main one. Only the condition related to the bounding of the algorithm is not natural to greedy algorithms (like list scheduling). A mechanism has to be implemented to get this property. The hypothesis about data dependencies (second one) is generally satisfied by software optimization systems, and of course the machines resources are limited.

5.2 Unrolling property

We are here able to give the important theorem, basis of the algorithm.

Theorem 5.6 *If the previous hypothesis are verified then $\Omega_{u^\infty(C)}$ is periodic. In other words there exist K, p et l such as:*

$$\forall j,\ \Omega_{u^\infty(C)}(MO_j^i) =$$

if $i < l$

$$\omega_{C_i}(MO_j^i)$$

and if $i \geq l$

$$((i-l) div\ K) * p + \omega_{C_{((i-l) mod K)+l}}(MO_j^{((i-l) mod K)+l})$$

p is the latency between two patterns in the scheduling:

$$p = first\_cycle(\omega_{C_{l+K}}) - first\_cycle(\omega_{C_l})$$

K ($K > 0$) is the number of loop bodies belonging to the scheduling pattern. l is the number of unrolling to be performed before the pattern is found.

Definition 5.7 *Two scheduling are said to be equivalent iff:*

$$\exists c \in N,\ \forall k,\ \omega_{C_i}(MO_k^i) = c + \omega_{C_j}(MO_k^j)$$

That we note:

$$\omega_{C_i} \equiv \omega_{C_j}$$

To prove the theorem 5.6 we use the following lemma:

Lemma 5.8 *If there exist* $\omega_{C_i} \equiv \omega_{C_j}$ *and* $MS(\Omega_{u^i(C)}) \equiv MS(\Omega_{u^j(C)})$ *then* $\Omega_{u^\infty(C)}$ *is periodic.*

Proof

To demonstrate the lemma 5.8 we have to examine the initial state of the scheduling of an iteration, because the compaction algorithm is deterministic, and for the same initial state and the same microcode it gives the same scheduling. The scheduling of an iteration depends on:

1. the microcode of the iterations which is the same for all the iterations since no transformation is applied during the unrolling process.
2. the dependency graph: we supposed that it is the same for all the iteration.
3. the scheduling of the previous iteration.
4. the machine state: taking into account the previous hypothesis about the scheduling of iteration i, the machine state of this iteration only interferes with the machine state of iteration $i-1$, because of the property 5.4 on the global scheduling of the loop.

Accordingly if $\omega_{C_i} \equiv \omega_{C_j}$ and if the machine states are equivalent, the initial states are equivalent for the compaction algorithm, such that $\omega_{C_{i+1}} \equiv \omega_{C_{j+1}}$. By induction it is easy to prove that for all $n > i$, $\omega_{C_{i+n}} \equiv \omega_{C_{j+n}}$, because two identical scheduling contribute in the same way to the state of an iteration. If the lemma is verified then $\Omega_{u^\infty(C)}$ is periodic and the following pattern $\omega_{C_i}, ..., \omega_{C_{j-1}}$ occurs.

To prove the theorem 5.6 we just have to prove the existence of C_i and C_j with $\omega_{C_i} \equiv \omega_{C_j}$ and $MS(\Omega_{u^i(C)}) \equiv MS(\Omega_{u^j(C)})$. The compaction algorithm is bounded, so there is a finite number of scheduling. This implies that in $\Omega_{u^\infty(C)}$ there exists an infinity of equivalent scheduling. The machine states for the iterations are also bounded, so there are at least two such states that are identical

■

5.3 Algorithm

The algorithm is directly deduced from the theorem. Theorem 5.6 ensures that the algorithm converges and that we can construct a software pipelining from the scheduling pattern with latency p.

All the dependencies (loop carried and intra dependencies) are satisfied by this software pipelining, and the modulo constraint is respected, since by construction it is equivalent to the unrolling, and loop carried dependencies are taken into account by the compaction process. The algorithm is the following:

1. Input: The global dependency graph (GGD).
2. Output: The optimized loop.
3. Program:

```
Begin
    pattern_not_found = True;
    i = 1;
    While pattern_not_found Do
        Ω_{u^{i+1}(C)} = Schedule(u^{i+1}(C));
        If nil ≠ Period(u^{i+1}(C), Ω_{u^{i+1}(C)}) Then
          pattern_not_found = False;
        Endif
        i = i+1;
    Enddo
    Construct_software_pipelining(Period(u^{i+1}(C),
    Ω_{u^{i+1}(C)}));
    Add_last_iterations();
End

Procedure Period(u^{i+1}(C), Ω_{u^{i+1}(C)}))
    Begin
        For k = i to 1 Do
          If ω_{C_k} = ω_{C_{i+1}} and
          MS(Ω_{u^k(C)}) = MS(Ω_{u^{i+1}(C)})
            Then Return(ω_{C_k},...,ω_{C_{i+1}});
          Endif
        Enddo
        Return(nil);
    End
```

The main problem of the algorithm is to ensure that the compaction algorithm is really bounded. To this end, a mechanism whose goal is to restrain the length of the scheduling of the iterations has been implemented. It prohibits instructions of an iteration to be scheduled too early in the unrolled loop. It is important to note that this mechanism do not degrade the optimization since microoperations which can be scheduled earlier are not constrain by data dependencies and resource occupation.

Let us suppose that the loop has been unrolled and compacted i times. A new iteration has to be added. Let lc be the last cycle of the first i iteration. In order to bound the scheduling of iteration $i+1$ this one cannot start before cycle

$$lc - x$$

and we have

$$S \leq x + cst$$

cst is the length of the scheduling of an iteration when all resources are available. It is the case for the first iteration. **x** is a loop body dependent parameter, and is determined heuristically. There are two cases:

1. If **x** is too large then in some cases the pattern is found after a large number of unrolling.
2. If **x** is too small then the optimization will be bad, because of the limitation of the iteration overlap.

In practice **x** can be initialized with the duration of the first iteration scheduling counterbalanced by a coefficient greater than one. **x** is used in practice to limit the first cycle at which the scheduling of an iteration can begin with respect to the last cycle used by the previous iteration. For instance **x** = **0** prohibits concurrency between iterations.

6 Evaluation of the Unrolling Technique

This section is dedicated to the evaluation of the algorithm described in section 5. To assess the efficiency of the algorithm we have compare it, in the case of V-loop (section 4.1), to the classical software pipelining technique (section 4). Let us give a brief overview of the target processor first.

6.1 The target processor

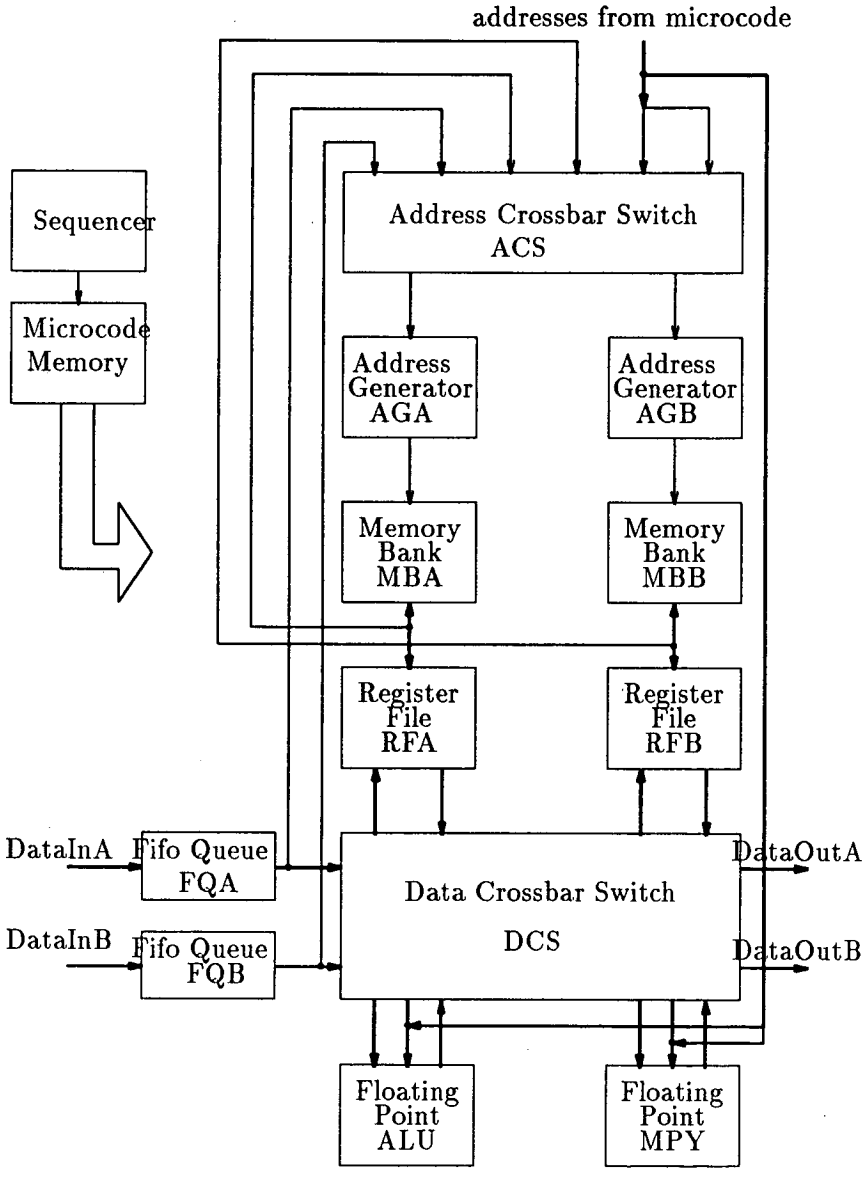


Figure 9: PCS processor data path

The PCS processor is the elementary operator of a linear array of processors to allow pipeline systems to be designed [4]. The array of processors is attached to a SUN-3 host workstation by means of a MC68020-based VME/VSB processor board and constitutes the workstation parallel accelerator. The PCS processors are linearly connected and the data flow is unidirectional. The processor executes single-precision floating-point operations at a peak rate of 20 MFLOPS. The processor organization

is depicted in Figure 9.

The processor architecture has been completely described with the machine model presented in section 2, complex functional units like the address generators have been taken into account.

6.2 Results

In this section we present preliminary results obtained on a single PCS processor. Two sets of benchmarks are presented. The first one is the 14-loop Livermore kernel (LLLx). The second one is a PCS kernel (PCSLx) extracted from applications running on the PCS processor (FFT...). The PCS kernel is essentially composed of loops which get data from the input queues. For instance PCSL 7 is the basic loop of a matrix multiplication mapped on a linear array of PCS processors where one matrix goes through the array. In order to present performances of loop optimization we use the Hockney formula [14]:

$$T(N) = \frac{(N + N_{1/2})}{V_\infty}$$

- N: is the number of iterations of the loop.
- $T(N)$: is the execution time for N iterations.
- V_∞: is the asymptotic speed (in MITS: Million iterations per second), which is the speed obtained when N is the infinity.
- $N_{1/2}$: is the number of iterations to get half of V_∞ ($V_{1/2}$).
- $v(N)$: is the speed in MITS for N and is equal to:

$$v(N) = \frac{N}{T(N)}$$

The two techniques (software pipelining and loop unrolling) have been compared using this formula, $N_{1/2}$ and V_∞ have been deduced. These values give a good overview of the performances obtained by such loop optimization algorithms: V_∞ gives the maximum achievable performance, $N_{1/2}$ is very closed to the number of overlapped iterations. If $N_{1/2}$ equal zero, there is no overlapping between the iterations, this is the case when a given iteration saturates machine resources, or if there are loop carried data dependencies between the first calculation of an iteration and the last result given by the previous one.

The Fortran programs were manually translated into the **lpcs** syntax, designed for programming PCS processor. The **lpcs** language is very close to the **W2** language [13] for the Warp processor. The translation was straightforward. LLL8 was just compacted because of the great number of operations (more than 300 microoperations). The performances of the loops are presented in Table 1.

In the case of the software pipelining, the results are only given for V-loops (however R-loops with loop-carried dependencies on registers are also treated). Results are close to optimal in the case of the unrolling technique. However, as expected, the use of this technique produces a higher value of $N_{1/2}$ than the classical software pipelining technique does.

The number of unrolling necessary to obtain a repetitive pattern of the scheduling is not very important (around 9 in average), however in the case of V-loops the method is expensive.

About register occupation, we have observe that the use of registers is better in the case of the unrolling method, that is due to the fact that the scheduling of an iteration takes into account the previous one.

loop	unrolling (l)	p	K	lat	$N_{1/2}$ (U)	V_∞ (U)	$N_{1/2}$ (S)	V_∞ (S)
LLL 1	14	16	3	5	6.7	9.3	4.2	10*
LLL 2	9	18	2	9	4.4	10*	3.78	10*
LLL 3	7	4	1	4	3.5	5*	1.7	5*
LLL 4	7	4	1	4	3.5	5*	1.7	5*
LLL 5	4	10	1	10	1.9	2*		
LLL 6	10	3	1	3	4.3	6.7*	2.25	5
LLL 7	10	18	1	18	3	8.9*	1.17	8.9*
LLL 8					0	4.5		
LLL 9	7	22	1	22	2.7	7.7*	0.93	6.3
LLL 10	11	31	1	31	2.9	2.5*	0	2.3
LLL 11	4	6	1	6	1.8	1.6*		
LLL 12	7	5	2	11	4.4	4	1.6	3.3
LLL 13	9	35	1	35	3.1	2.5		
LLL 14	22	107	4	26	6.9	4.2		
PCSL 1	17	12	2	6	4.1	6.6*	1.8	5.7
PCSL 2	11	4	1	4	2.7	5*	1.2	5*
PCSL 3	4	8	1	8	2	7.5*	1	7.5*
PCSL 4	6	2	1	2	3.5	5*	2.5	5*
PCSL 5	18	12	2	6	5.5	1.6*	1.6	1.6*
PCSL 6	7	5	2	3	4.8	12	2.3	10
PCSL 7	5	6	1	6	2.6	13.3*	1.1	11.4
PCSL 8	7	2	1	2	5	20*	4	20*
PCSL 9	8	26	2	13	2	7.6*	1.9	5.9
PCSL 10	7	6	1	6	3.1	10*	2	10*
PCSL 11	7	6	1	6	3.1	10*	1.8	10*

Table 1: Livermore loops and PCS kernel

Table 1 description:

- The rates in the columns are given in MFLOPS.
- unrolling: number of unrolling needed to find the repetitive pattern.
- p: latency between two patterns.
- K: number of loop bodies in the pattern.
- Lat: latency between successive iterations which is equal to the ratio p/K (the result has been rounded).
- V_∞ (U) gives the asymptotic speed obtain by our algorithm, and $N_{1/2}$ (U) gives the number of iteration necessary for achieving half of this asymptotic speed.
- V_∞ (S) gives the asymptotic speed obtain by the classical software pipeline algorithm, and $N_{1/2}$ (S) gives the number of iteration necessary for achieving half of this asymptotic speed.
- Optimal values of V_∞ are marked with (*).

References

[1] A. V. Aho and J. D. Ullman. *Principles of Compiler Design.* Addison-Wesley, 1977.

[2] A. Aiken and A. Nicolau. A development for horizontal microcode programs. *MICRO 19*, 23–31, 1986.

[3] A. Aiken and A. Nicolau. Optimal loop parallelization. *Proceedings of the SIGPLAN '88*, 308–317, 1988.

[4] F. Bodin, F. Charot, and C. Wagner. Overview of a high-performance programmable pipeline architecture. *ACM Supercomputing 89 (Crete)*, 398–409, 1989.

[5] S. Dasgupta and J. Tartar. The identification of maximal parallelism in straight-line microprograms. *IEEE Transactions on Computers*, 25(10):986–991, 1976.

[6] S. Davidson, D. Landskov, B. D. Shriver, and P. W. Mallett. Local microcode compaction techniques. *Computing Survey*, 12(3):261–294, 1980.

[7] D.J. Dewit. *A Machine Independent approach to the Production of Horizontal Microcode.* PhD thesis, University of Michigan, 1976.

[8] C. Eisenbeis. Optimisation automatique de programmes sur array-processors. Thèse d'université de Pierre et Marie Curie Paris 6, Juin 1986.

[9] C. Eisenbeis. Optimization of horizontal microcode generation for loop structures. *ACM Supercomputing 88*, 453–465, 1988.

[10] C. Eisenbeis, W. Jalby, and A. Lichnewsky. Squeezing more cpu performance out of a cray-2 by vector block scheduling. *Florida Supercomputing 88*, 1988.

[11] J.A. Fisher. Trace scheduling: a technique for global microcode compaction. *IEEE Transactions on Computers*, 30(7):478–490, 1981.

[12] M. R. Garey and D.S. Johnson. *Computers and Intractability, A Guide to the Theory of NP-Completeness.* W.H. Freeman and company, 1979.

[13] T. Gross and M. S. Lam. Compilation for a high-performance systolic array. *SIGPLAN'86 Symposium on Compiler Construction*, 27–38, 1986.

[14] R.W. Hockney and C.R. Jesshope. *Parallel Computers.* Adam Hilger Ltd, Bristol, 1981.

[15] R.L. Kleir and C.V. Ramamoorthy. Optimization strategy for microprograms. *IEEE Transactions on Computers*, 20(7):783–794, 1971.

[16] P. M. Kogge. *Architecture of Pipelined Computers.* Mc Graw-Hill, 1981.

[17] M. Lam. *A Systolic Array Optimizing Compiler.* PhD thesis, Carnegie Mellon University, May 1987.

[18] R.A. Mueller and V.H. Allan. Compaction with general synchronous timing. *IEEE Transactions on Software Engineering*, 595–599, 1988.

[19] A. Nicolau. *A Fine-Grain Parallelizing Compiler.* cornell university, ithaca new york edition, December 1986.

[20] J.H. Patel and E.S. Davidson. Improving the throughput by insertion of delays. *Proc 3rd Annual Symp. on Computer Architecture*, 159–164, 1976.

[21] B.R. Rau and C.D. Glaeser. Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing. *IEEE*, 183–198, 1981.

[22] R.F. Touzeau. A fortran compiler for the fps-164 scientific computer. *Proc. of the ACM SIGPLAN '84 Symp. on Compiler Construction*, 48–57, 1984.

LISTE DES DERNIERES PUBLICATIONS INTERNES

PI 506 **VM pRAY, AN EFFICIENT RAY TRACING ALGORITHM ON DISTRIBUTED MEMORY PARALLEL COMPUTER**
Didier BADOUEL, Thierry PRIOL
Janvier 1990, 50 Pages.

PI 507 **ON THE REGULAR STRUCTURE OF PREFIX REWRITINGS**
Didier CAUCAL
Janvier 1990, 32 Pages.

PI 508 **RAY TRACING ON DISTRIBUTED MEMORY PARALLEL COMPUTERS: STRATEGIES FOR DISTRIBUTING COMPUTATIONS AND DATA.**
Didier BADOUEL, Kadi BOUATOUCH, Thierry PRIOL
Janvier 1990, 16 Pages.

PI 509 **STABILITY ANALYSIS AND IMPROVEMENT OF THE BLOCK GRAM-SCHMIDT ALGORITHM**
William JALBY, Bernard PHILIPPE
Janvier 1990, 24 Pages.

PI 510 **TESTING FOR THE UNBOUNDEDNESS OF FIFO CHANNELS IN PROGRAMS.**
Thierry JERON
Janvier 1990, 30 Pages.

PI 511 **AUTOMATIC ANIMATION CONTROL OF PHYSICAL SYSTEMS.**
Georges DUMONT, Bruno ARNALDI, Gérard HEGRON
Janvier 1990, 22 Pages.

PI 512 **A FAULT TOLERANT TIGHTLY COUPLED MULTIPROCESSOR ARCHITECTURE BASED ON STABLE TRANSACTIONAL MEMORY**
Michel BANATRE, Philippe JOUBERT
Février 1990, 20 Pages.

PI 513 **BUILDING A GLOBAL TIME ON PARALLEL MACHINES**
Jean-Marc JEZEQUEL
Février 1990, 28 Pages.

PI 514 **PARALLELISATION D'UN RESEAU NEURONAL**
Krysztof WOLINSKI
Février 1990, 20 Pages.

PI 515 **UNE NOUVELLE APPROCHE DE LA RELATION VISION-COMMANDE EN ROBOTIQUE**
Bernard ESPIAU, François CHAUMETTE, Patrick RIVES
Février 1990, 46 Pages.

PI 516 **COMMENT INTRODUIRE LA CONTIGUITE EN ANALYSE DES CORRESPONDANCES ? Application en segmentation d'image.**
Brigitte ESCOFIER, Habib BENALI, Kaddour BACHAR
Février 1990, 26 Pages.

PI 517 **MACHINE MODELING AND LOOP OPTIMIZATION FOR HORIZONTAL MICROCODED MACHINES**
François BODIN, François CHAROT
Février 1990, 24 Pages.

Imprimé en France
par
. l'Institut National de Recherche en Informatique et en Automatique .